

Implementing Access Control Markov Decision Processes with GLPK/GMPL^{*}

Charles Morisset

IIT-CNR, Security Group
Via Giuseppe Moruzzi 1, 56124 Pisa, Italy,
`charles.morisset@iit.cnr.it`

Abstract. In a recent approach, we proposed to model an access control mechanism as a Markov Decision Process, thus claiming that in order to make an access control decision, one can use well-defined mechanisms from decision theory. We present in this paper an implementation of such mechanism, using the open-source solver GLPK, and we model the problem in the GMPL language. We illustrate our approach with a simple, yet expressive example, and we show how the variation of some parameters can change the final outcome. In particular, we show that in addition to returning a decision, we can also calculate the value of each decision.

1 Introduction

An access control mechanism is responsible within an information system of intercepting any access made by a user over a resource, deciding whether this access should be allowed or not, and enforcing the corresponding decision. Most existing mechanisms rely on the definition of an *access control policy*, defined in a dedicated language (e.g., RBAC [7], XACML [15]), which can be roughly seen as a set of rules separating the set of accesses into the secure ones and the non-secure ones.

We recently proposed in [13] to model an access control mechanism as a Markov Decision Process (MDP) [2]. The main strength of this approach is to account for uncertainty and probabilistic behaviour of the system directly into the decision mechanism. From a global point of view, such a modelling expresses the fact that making an access control decision can be done using tools from decision theory. By using an MDP instead of a traditional security mechanism, one can express an intuitive notion of *utility* rather than a standard classification between secure and non-secure situations. This is particularly useful in some critical environments, where at a given point in time, it might only be possible to choose between “bad” situations.

For instance, a study [17] revealed that in a particular healthcare system, 74% of users have manual overriding permissions and 54% of records have been accessed at least once using an overriding permission. Such situations are due

^{*} This work is supported by the EU project NESSoS.

to the conflicting choice between respecting the static policy and providing the best possible care to the patients. The typical example is that of a nurse, who is normally not authorized to access the medical record of a patient, and who needs to access it in order to deliver some treatment while no attending physician is available. The information system can only choose between two “bad” options: granting the access to the nurse, and therefore breaching patient confidentiality, or denying the access, and therefore risking the life of the patient.

We present in this paper a detailed example of the implementation of an Access Control-Markov Decision Process (AC-MDP) using GLPK/GMPL. The objective of this paper is therefore to present a basic and simple guide of what to do and what to expect when implementing an AC-MDP. In order to do so, we introduce a simple running example, that we will tweak along the paper in order to illustrate how changing some parameters might change the results. We try to present our approach from the perspective of a security engineer responsible for implementing a security mechanism for a concrete problem, loosely inspired from the healthcare context. The rest of this paper is therefore organized as follows: in Section 2, we state the concrete problem we want to implement and we recall the basic definition of an AC-MDP. In Section 3, we present the general implementation in GLPK/GMPL of the AC-MDP for this concrete problem, and we instantiate this general model in Section 4. Finally, we discuss the different results and the important points to extract from our approach in Section 5.

Related Work

To the best of our knowledge, our recent approach [13] was the first usage of Markov Decision Processes in the context of access control systems. However, the problem of dealing with risk and uncertainty for access control systems has been already studied in the literature. For instance, Aziz *et al.* [1] refine a policy to a more restrictive one, in order to deal with threats.

Risk is often considered as an input to the system, that must stay below a certain threshold. Cheng *et al.* introduce in [4] the Fuzzy Multi-Level Security model, where each access is associated with a level of risk, and the final decision of the authorization mechanism is given according to some predefined risk thresholds. Diep *et al.* extend this approach in [6] by considering costs in terms of availability, integrity and confidentiality for each decision, and use thresholds for each corresponding risk.

Some approaches aim at calculating the risk from the environment. For instance, Ni *et al.* introduce in [16] fuzzy security parameters that can be inferred from traditional parameters, hence introducing a notion of uncertainty directly into the parameters of the system, while Chen and Crampton present in [3] a way to calculate the risk for a RBAC model, using intuitive notions of competence and distance between users, roles and permissions. The probabilistic change of security attributes is considered by Krautsevich *et al.* in [10], who model this change using a Markov chain.

Dealing with uncertainty requires quantitative techniques, and therefore accesses must be associated with a utility value. Some models use these notions,

for instance Krautsevich *et al.* extend in [11] the auto-delegation mechanism [5] with probabilistic availability, reusing some notions of utility functions previously introduced in [9]. Similarly, Molloy *et al.* present in [14] a model to predict and make local decisions under uncertainty, where the system needs to choose between taking a decision locally or defer it to a central server, according to the utility of the access and the cost of communicating with the server.

Beyond the scope of access control, several pieces of work use the concept of Markov Decision Process (MDP) in the context of security. For instance, Kreidl introduces in [12] a simple MDP with only three states (normal, under attack and failure) and three decisions (wait, defend and reset), which analyses the cost of defending countermeasures against the cost of an intrusion.

Similarly, He *et al.* present in [8] an analysis of the operational costs and the negative and positive impact of security countermeasures using Domain Partitional Markov Decision Processes, which partition the network into several security domains, each domain coming with its own MDP. This work, as the previous one, mostly focuses on the detection of intrusions and the decisions needed when some are discovered.

Finally, Singh *et al.* use in [18] an MDP to make channel assignments for network devices. This problem can be considered as a special instance of an access control problem, where the devices ask to access channels, with a specific policy stating that any device can access a channel, but the more devices use a given channel, the lower is the utility for this channel. Hence, our approach can be seen as a more general approach, where the policy is not constrained.

2 Problem Statement

As we said in the Introduction, we present our approach from the perspective of security engineer, who is responsible for implementing a security mechanism for a concrete problem. Let us first introduce this problem.

Example 1 *Let S be a simple system such that:*

1. *There are two users, **alice** and **bob**, such that **alice** (for instance, a physician) is more qualified than **bob** (for instance, a nurse);*
2. *There are two resources, **high** and **low**, such that the resource **high** (for instance, one of **alice**'s patient record) is more sensitive than **low** (for instance, some indicates related to a drug);*
3. ***bob** is not normally qualified enough to access **high**;*
4. *in case of emergency (for instance, the patient is having a heart attack), the resource **high** should be accessed.*

The role of the security engineer is therefore to define a mechanism that, given some the current state and an access (u, r) , where u is a user and r a resource, returns a security decision, such as **allow** or **deny**. The traditional approach to do so is to define a security policy, that is, a set of rules describing what decision

to make for each access, and then to define a simple program that checks if the given access satisfies the policy or not. For instance, we could define the following policy, for any access (u, r) :

If $u = \mathbf{bob}$, $r = \mathbf{high}$ and there is no emergency, then **deny** else **allow**.

There exist many languages (e.g., XACML) in which this policy can be defined. However, defining a “static” policy requires to resolve beforehand all possible situations. In our example, there is a clear conflict between rules 3 and 4: in case of emergency, if **alice** is not accessing **high**, then either we allow **bob** to access it, thus breaking rule 3, or we do not, thus breaking rule 4. Note that there exist some approaches to deal with this particular problem of delegation (in case of unavailability, **alice** should automatically delegate her right over **high** to **bob**), such as [5,11], however we want to illustrate a more general problem: the combination of rules might lead to conflicting situations. The resolution of such conflicts is usually addressed by considering composition operators, thus allowing some rules to take precedence over others. We propose here a different approach, where conflict resolution is obtained as the result of an optimization process.

Indeed, we defined in [13] a novel approach, where the security policy is not defined, but *derived* from a decision process. Intuitively, the responsibility of the security engineer is only to put some values on the different resource and/or states of the system, to describe the probabilistic behaviour of the system, and the optimal policy can be automatically defined. More precisely, the security engineer must define an Access Control Markov Decision Process (AC-MDP), given by:

Definition 1 *An AC-MDP is a tuple $\langle \Sigma, \mathcal{A}, \mathcal{P}, \mathcal{W} \rangle$, where:*

- $\Sigma = \mathcal{I} \times \mathcal{R}$ is a set of access control states, such that \mathcal{I} represents the security information of the state and \mathcal{R} the access requests (each state containing a request to control),
- \mathcal{A} is a set of decisions,
- $\mathcal{P} : \Sigma \times \mathcal{A} \times \Sigma \rightarrow [0, 1]$ is the probability function, such that $\mathcal{P}(\sigma_i, a, \sigma_j)$, which, for the sake of exposition, we also write p_{ij}^a , stands for the probability of reaching the state σ_j by executing the decision a from the state σ_i ,
- $\mathcal{W} : \Sigma \times \mathcal{A} \times \Sigma \rightarrow \mathcal{U}$ is the reward function, such that $\mathcal{W}(\sigma_i, a, \sigma_j)$, also written as w_{ij}^a , stands for the reward associated with executing the decision a from the state σ_i and arriving in the state σ_j .

It is worth noting that we consider here that the reward is a *gain* function, and that therefore the objective of the process is to eventually *maximize* the accumulated rewards. We could equivalently consider the reward as a *cost*, and in this case the objective would be to minimize it. In the following, for the sake of clarity, we associate gain with positive rewards, and cost with negative rewards, the important point being to *compare* two values between each other.

We write q_i^a for the immediate reward for executing the decision a in the state σ_i :

$$q_i^a = \sum_j p_{ij}^a \cdot w_{ij}^a \quad (1)$$

In this context, a security policy is a function $\delta : \Sigma \rightarrow \mathcal{A}$, and the value V^δ of each state for this policy can be defined as:

$$V^\delta(\sigma_i) = q_i^{\delta(\sigma_i)} + \beta \sum_j p_{ij}^{\delta(\sigma_i)} V^\delta(\sigma_j) \quad (2)$$

where $0 \leq \beta \leq 1$ is a discount factor, indicating how much weight is put on the value of future states. The optimal policy δ^* is the policy that maximizes the value function, that is, for any policy δ and any state σ , we have $V^{\delta^*}(\sigma) \geq V^\delta(\sigma)$. More formally, given a state σ_i , the optimal policy δ^* is given by:

$$\delta^*(\sigma_i) = \arg \max_{a \in \mathcal{A}} [q_i^a + \beta \sum_j p_{ij}^a V^{\delta^*}(\sigma_j)] \quad (3)$$

Clearly, this policy is not necessarily unique. For instance, in the degenerate case where all decisions are associated with the same reward, then any policy is optimal. This case would be equivalent to defining a policy for an access that has the same impact whether it is allowed or denied. In order to define the actual policy and to solve the non-determinism of choosing between several optimal policies, we could introduce an arbitrary ordering among decisions, such that if in a given state, two decisions maximize the value, we take the smallest.

With our approach, instead of defining directly the policy, the security engineer instantiates the AC-MDP (and in particular the function \mathcal{P} and \mathcal{W}), and the optimal policy can be calculated as an optimization problem. We present in the following sections how to perform such instantiation for the concrete example.

3 Definition of the AC-MDP in GMPL

GLPK¹ is a piece of software intended for solving large-scale linear programming problems, and it supports the GMPL language for modeling problems, which is a subset of AMPL. We now show how to implement an AC-MDP in GMPL. For the sake of exposition, we first present the “model” part of the implementation, that is, the part that does not depend on concrete values for the parameters, and we define in the following section some concrete values for these parameters. This implementation is directly inspired from that provided by Vincent Conitzer in his lecture on Linear and Integer Programming².

¹ Open source software available at <http://www.gnu.org/software/glpk/>

² <http://www.cs.duke.edu/courses/spring08/cps296.2/>

3.1 States and Actions

We first define the set \mathcal{I} of security information and the set \mathcal{R} of requests. Based on the problem description, we assume the qualifications of the entities to be fixed, thus not belonging to the state³, and therefore we define the security information as a pair (e, c) , where e is a boolean indicating whether there is a current emergency and c is the set of current accesses, such that an access is simply a pair (u, r) . Furthermore, we model a request directly as an access.

For the sake of generality, we define as parameters the number of users NU and resources NR , and we define the set of users and resources as indices. We also introduce the “empty” request (eps, eps) , that represents the fact that there is no access request to control.

```
param NU default 2;
set USERS := 0..(NU-1);

param NR default 2;
set RESOURCES := 0..(NR-1);

set ACCESS := USERS cross RESOURCES;

param eps, symbolic;
set FACCESS := ACCESS union {(eps, eps)};
```

The state should include all previous currently granted accesses, however GMPL does not natively support powersets. Instead, we use a workaround by indexing each element of the powerset: we first associate each access (u, r) with the index $2^{u \cdot \text{NR} + r}$ (e.g., in our settings, the index of $(0, 0)$ is 2^0 , the index of $(1, 1)$ is $2^{1 \cdot 2 + 1} = 2^3$, etc); we then create the set PA , which ranges from 0 to $2^{\text{NU} \cdot \text{NR}} - 1$; finally, we define the set POW which is indexed by PA , such that, intuitively speaking, in the binary representation of k , the i -th bit is equal to 1 if, and only if, $\text{POW}[k]$ contains the access indexed by i . For instance, for the index $k = 5$, that is $k = 101$ in binary, we have that $\text{POW}[5]$ contains exactly the accesses indexed by 2^0 and 2^2 , i.e., the accesses $(0, 0)$ and $(1, 0)$. This encoding is defined in GMPL as follows.

```
param n := (NU)*(NR);
set PA := 0 .. (2**n - 1);
set POW {k in PA} := setof{(u,r) in ACCESS: (k div 2**(u * NR + r)) mod 2 = 1}(u,r);
```

Note that the indexation of POW is done over ACCESS and not FACCESS , meaning that, by construction, the empty request cannot belong to the set of previously granted accesses. A state is then given by an emergency status, a set of accesses and an access to control:

```
param calm, symbolic;
param alert, symbolic;
set EMERGENCY := {calm, alert};

set STATES := EMERGENCY cross PA cross FACCESS;
```

³ It is of course possible to include qualification functions in the state, but that would make its structure more complex, with no real gain for the concrete example.

For instance, the state `[calm, 1, 1, 0]` represents the state of the system that is in a `calm` status, where the access `[0,0]` (corresponding to `POW[1]`) has been previously granted and where the current access to control is `[1,0]`⁴. Finally, we define the set of actions $\mathcal{A} = \{\text{allow}, \text{deny}\}$.

```
param allow, symbolic;
param deny, symbolic;

set ACTIONS := {allow, deny};
```

3.2 Transition function

We now show how to define the transition function \mathcal{P} , using three sub-functions, one for each component of a state. The probability of switching from one emergency status to another is given as a concrete parameter (we actually study the behaviour of the model when this parameter is varying in Section 4.3) and therefore only declared in the model.

```
param transition_emergency {e1 in EMERGENCY, e2 in EMERGENCY};
```

Concerning the set of current accesses, for the sake of simplicity, we do not consider here probabilistic modifications. In other words, whenever an access is allowed, the following state contains it. However, it would be straightforward to extend this transition function to also consider the possibility that an access granted is not added, or even that a denied access is in fact performed. We could also model here the fact that at any time, there is a risk of leakage of information, that is, there is a non-null probability that a non requested access will be added to the set of current accesses in the next state.

```
param transition_access {s1 in PA, (u, r) in FACCESS, a in ACTIONS, s2 in PA} :=
  if (a = deny or u = eps) then (if s1 = s2 then 1 else 0)
  else (if ((POW[s2] within (POW[s1] union {(u, r)}))
    and ((POW[s1] union {(u, r)}) within POW[s2])) then 1 else 0);
```

An important point of our modelling is the fact that the state contains the request to control, and it follows that when defining the transition to the next state, we must also define what will be the next request to control. We first assume that it is always `[eps, eps]` (i.e., we only control one request at the time), and we release this assumption in Section 4.2.

```
param transition_req {s1 in PA, (u, r) in FACCESS, a in ACTIONS, (u2, r2) in FACCESS} :=
  if (u2 = eps and r2 = eps) then 1 else 0;
```

Finally, the transition probability is given by the product of the three previous sub-functions.

```
param transition {(e1, s1, u1, r1) in STATES, a in ACTIONS, (e2, s2, u2, r2) in STATES} :=
  transition_emergency[e1, e2] * transition_access[s1, u1, r1, a, s2]
  * transition_req[s1, u1, r1, a, u2, r2];
```

⁴ Note that in GMPL, there is an implicit “inlining” of the cartesian product, that is, in the above example, we do not write the state as `[calm, 1, [1, 0]]`, even though the access `[1, 0]` is a pair itself.

3.3 Reward Function

To some extent, defining the reward function corresponds to defining the policy, in the sense that instead of stating if an access is correct or not, the security engineer attaches a value to it, and this value will be later use to determine whether the access should be allowed or not. Hence, we declare to basic reward functions: `reward_access[u,r]` indicates the value gained by granting the access `[u,r]`, and `reward_resource[r]` indicates the value of not accessing the resource `r` in case of emergency. We instantiate these functions in Section 4.

```
param reward_access {(u, r) in ACCESS};
param reward_resource {r in RESOURCES};
```

From the function `reward_resource`, we can define the reward associated with a state, which is, in case of emergency, the sum of the reward of each resource.

```
param reward_emresource {(e, s, u, r) in STATES} :=
  if (e = calm) then 0 else
  sum {r1 in RESOURCES: forall{u1 in USERS} (u1,r1) not in POW[s]} reward_resource[r1];
```

Finally, we can define the reward associated with an entire transition, thus corresponding to the function \mathcal{W} of the AC-MDP.

```
param reward_transition {(e, s, u, r) in STATES, a in ACTIONS, (e2, s2, u2, r2) in STATES} :=
  (if u = eps and r = eps then 0) else
  (if a = allow then reward_access[u,r] else 0) + (reward_emresource[e2,s2,u2,r2]);
```

It is worth noting that if the state contains the empty request to control, then regardless of the decision, the reward of the transition is null.

3.4 Value Function and Policy

We can now define the optimization problem, in order to calculate the maximal value function, and thus the optimal policy. We first introduce the immediate reward, as defined in Equation (1).

```
param immediate_reward {(e, s, u, r) in STATES, a in ACTIONS} :=
  sum {(e2, s2, u2, r2) in STATES} (transition[e,s,u,r,a,e2,s2,u2,r2]
    * reward_transition[e,s,u,r,a,e2,s2,u2,r2]);
```

Intuitively, we want to define the value function of the optimal policy V^{δ^*} , that is, for each state σ_i , we want to define the equation:

$$V^{\delta^*}(\sigma_i) = \max_{a \in \mathcal{A}} [q_i^a + \beta \sum_j p_{ij}^a V^{\delta^*}(\sigma_j)]$$

in which case we would just need to solve the corresponding system of equations. However, we cannot directly follow this approach, because the operator \max is not linear. As stated by Conitzer in his lecture⁵, the typical solution for this kind of problem is to define the equations such that $V^{\delta^*}(\sigma_i)$ must be greater or equal to $q_i^a + \beta \sum_j p_{ij}^a V^{\delta^*}(\sigma_j)$, for any decision a , and to minimize the value of $V^{\delta^*}(\sigma_i)$, which thus corresponds to the lowest greater bound. In order to minimize each $V^{\delta^*}(\sigma_i)$, we simply aim at minimizing the sum of all the values.

⁵ <http://www.cs.duke.edu/courses/spring08/cps296.2/applications.pdf>


```

var value{(e,s,u,r) in STATES};

minimize total: sum{(e,s,u,r) in STATES} value[e,s,u,r];
s.t. bellman{(e,s,u,r) in STATES, a in ACTIONS}: value[e,s,u,r] >=
    immediate_reward[e,s,u,r,a]
    + sum{(e2, s2, u2, r2) in STATES}(beta*transition[e,s,u,r,a,e2,s2,u2,r2]*value[e2,s2,u2,r2]);

```

4 Instantiation

The model presented in the previous section represents a “general” implementation of the problem the security engineer wants to address: the number of users and resources can easily be extended, the rewards for accesses and states are not precisely defined, the probability of changing the emergency status is not specified, neither is the discount factor β . We show in this section how to define these particular values, and we describe the corresponding results.

In order to compare the different results obtained, we focus on the value, from the empty state (i.e., where no access has been previously granted), of allowing or denying each access. More precisely, given an emergency status e and an access (u, r) such that $\sigma_i = (e, \emptyset, (u, r))$, and a decision a , we define the decision value DV as:

$$DV(e, (u, r), a) = q_i^a + \beta \sum_j p_{ij}^a V^{\delta^*}(\sigma_j)$$

The security mechanism then selects the decision with the highest value.

4.1 Basic Instantiation

We first define a “naive” instantiation, somewhat equivalent to the static policy presented in Section 2. In order to remove the values of the future states from the equation, we set the discount factor to 0. We also assume that the emergency status cannot change (i.e. we should make the decision considering that the status will not change).

```

param transition_emergency:= [*,*]:
    calm  alert  :=
calm  1    0
alert 0    1;

```

We also give arbitrary values for the reward function, trying to keep the idea of the original policy (for the sake of readability, we have used the names of the users and resources instead of their indices, as it is done in the actual implementation).

```

param reward_access := [*,*]:
    high  low  :=
alice  6    10
bob    4    -10;

param reward_resource := low 0 high -20;

```

Table 1. Decision values for $\beta = 0$ and emergency probability of 0

Status	Decision	(alice, low)	(alice, high)	(bob, low)	(bob, high)
Calm	deny	0	0	0	0
	allow	6	10	4	-10
Alert	deny	-20	-20	-20	-20
	allow	-14	10	-16	-10

As for the policy defined in Section 2, we consider here that the reward for leaving the resource **high** non accessed in case of emergency is worse than letting **bob** accessing **high**. However, as we will see in the following, this does not necessarily imply that **bob** will always be able to access **high**.

Since the discount factor β is set to 0, the decision value is directly equal to the immediate reward. For instance, if we allow the access (**alice, low**), we get an immediate reward of 6, while allowing the access (**bob, high**) brings a reward of -10. Furthermore, if the reached state is in an emergency status, and the resource **high** is not accessed, we also get a reward of -20.

We present in Table 1 the decision values for these basic parameters. The only situation where the decision value of **deny** is higher than that of **allow** is when **bob** wants to access **high** in a **calm** status. It is however worth noticing that when there is an alert, both decision values for the access (**alice, low**) are negative, meaning that both decisions are “bad” (since **high** will not be accessed whatever decision is made), but the decision **allow** is still better than **deny**. This is an interesting aspect of quantitative approaches, as they enable the system to make the “best” decision, even though this decision might lead to a “bad” situation.

4.2 Complex Requests

The previous instantiation considers a very “static” configuration of the system, since the emergency status is not assumed to dynamically change and the discount factor is null. In practice, a system is not frozen in time, and does not stop after controlling one request. Hence, in order to make the “best” decision, it is necessary to take into account the possible future evolutions of the system. In the following, in addition to setting the discount factor to a non null value, we also consider two parameter variations: there is a non-null probability that a state in a calm status will transition to a state in an alert status, and another request might need to be controlled after the current one. In order to specify the former possibility, we define:

```
param transition_emergency:= [*, *]:
    calm    alert  :=
calm  0.9    0.1
alert 0      1;
```

Table 2. Decision values for $\beta = 0.9$ and emergency probability of 0.1

Request	Status	Decision	(alice, low)	(alice, high)	(bob, low)	(bob, high)
unique	Calm	deny	-2	-2	-2	-2
		allow	4	10	2	-10
	Alert	deny	-20	-20	-20	-20
		allow	-14	10	-16	-10
once	Calm	deny	2.63	2.63	2.63	2.63
		allow	7.58	14.15	6.41	-1.59
	Alert	deny	-23.54	-23.54	-23.54	-23.54
		allow	-15.54	14.15	-16.70	-1.59
all	Calm	deny	34.80	34.80	34.80	34.80
		allow	40.80	55	38.80	35
	Alert	deny	4.55	4.55	4.55	4.55
		allow	10.55	55	8.55	35

We then characterize the set of requests that are possible after executing a given action over a given request from a given set of accesses. More specifically, we distinguish between three cases⁶:

- **unique**: only one request is controlled, meaning that the only possible following request is the empty request.
- **once**: each request can only be allowed once.
- **all**: all non-empty requests are possible (simulating an infinite trace):

We present in Table 2 the decision values obtained by using the probability of change of the emergency status from **calm** to **alert** to 1.

It is worth observing that the request transition has an important impact on the actual decision values, but the ordering is almost always preserved. Indeed, the decision value of **allow** is always higher than that of **deny** for the accesses (**alice, low**), (**alice, high**) and (**bob, low**), and for the access (**bob, high**) when the state is known to be in the **alert** status. However, from a state in a **calm** status, for the behaviours **once** and **unique**, it is better to deny the access (**bob, high**) while for the behaviour **all**, it is better to allow it.

This result is somewhat counter-intuitive, since one could expect that for **once** and **unique**, if we deny the access, the probability that the resource **high** will be accessed is quite low, while for **all**, even if we deny the access to **bob**, we know that **alice** will ask for the access in the future, and therefore that **high** will be eventually accessed. In fact, this result is due to the fact that we automatically associate the state containing the empty request with a null value. In other words, we do not take into account the fact that the system might stay forever in a state where **high** is not accessed. On the other hand, for **all**, we have somehow infinite sequences of requests, which automatically gives a high value to all states.

⁶ For the sake of readability, we skip the implementation details. All the different GMP programs presented here can be found at <http://www.morisset.eu/qasa12/>

We can modify the behaviour such that starting from state `a` containing the empty request takes the reward of the reached state (i.e., in our example, the same state) into account:

```
param reward_transition {(e, s, u, r) in STATES, a in ACTIONS, (e2, s2, u2, r2) in STATES} :=
  (if a = allow and u != eps then reward_access[u,r] else 0)
  + (reward_emresource[e2,s2,u2,r2]);
```

In this case, the decision values for the **all** behaviour do not change, on the contrary of the **unique** and **once** behaviours, for which the ordering of the decisions for the access (**bob, high**) in the `calm` status actually change. Indeed, for **unique**, denying this access leads to a value of -105.26, while allowing it leads to -10. Similarly, for **once**, denying it leads to a value of -32.35, while allowing it leads to -1.59. In other words, the values for allowing the accesses are identical to those in Table 2, but the values for denying are much lower, thus reflecting the idea that the state is likely to remain in a “bad” state, where **high** is not accessed.

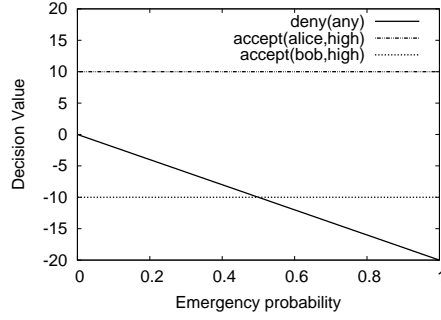
Nonetheless, it is interesting to observe that although the ordering between the different behaviors is usually the same, meaning that we make decisions consistent with the original policy, the difference between the value of each decisions can significantly vary. For instance, consider the access (**alice, low**) for the behaviour **all** in the `calm` status: denying it leads to a value of 34.80, while allowing it “only” leads to a value of 40.80. In other words, the gain of allowing versus denying it is quite low, relatively and absolutely speaking. If we were to attribute a priority for the treatment of this access, we could probably give it a low one.

On the other hand, for **all**, from the `alert` status, the differential for the access (**alice, high**) is very important: the value of denying it is 4.55, while allowing it brings a value of 55, i.e., multiplying the value by a factor greater than 10. In other words, there is no doubt at all that this access should be allowed in these conditions. It is therefore interesting to observe that in addition to the decision, our approach also provides the “strength” or “confidence” in the decision.

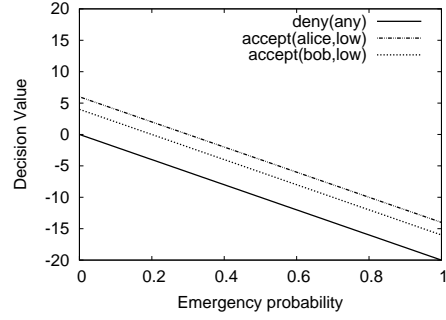
4.3 Variation of the emergency probability

Finally, our last experiment consists in calculating how, for a given discount factor ($\beta = 0.9$), the decision values vary when the emergency probability varies. Intuitively, if the system is very stable (i.e., the probability equals 0), then any transition from a `calm` state leads to a `calm` state, and the system does not need to care whether **high** is accessed or not. On the other hand, if the system is very unstable (i.e., the probability equals 1), then any transition will lead to an emergency state, in which case the system has to ensure as much as possible that the resource **high** is accessed.

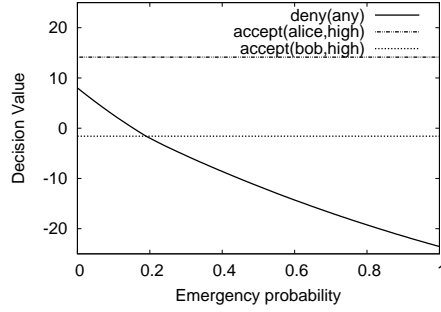
We show in Figure 1 these results, for each request behaviour. Firstly, we can observe that for the resource **low**, the value of denying an access to it, regardless of the user, is always lower than allowing it. It therefore fits with the intuition that whatever the emergency status, there is no gain in denying an access to the



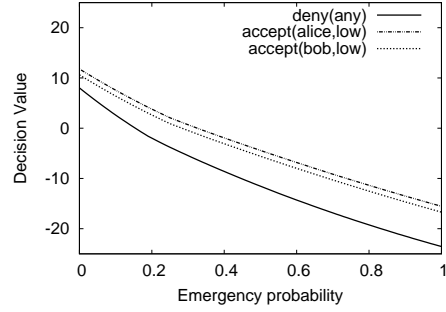
(a) Decision values for **unique,high**



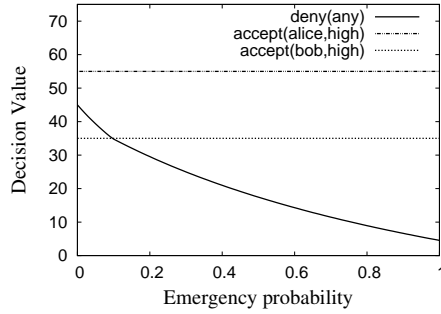
(b) Decision values for **unique,low**



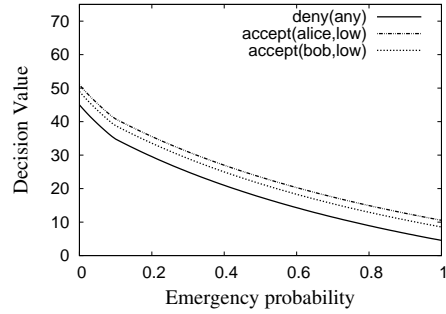
(c) Decision values for **once,high**



(d) Decision values for **once,low**



(e) Decision values for **all,high**



(f) Decision values for **all,low**

Fig. 1. Decision values for $\beta = 0.9$, where the emergency probability stands for `transition_emergency[calm,alert]`

resource **low**. We can similarly observe that for a given resource, the value of allowing the access for **alice** is always greater than allowing the access for **bob**, which is consistent with the requirement that **alice** is more qualified than **bob**.

Finally, it is interesting to see that the emergency probability required for the value of allowing (**bob, high**) to be higher than that of denying it changes according to the request behaviour: it is exactly 0.5 for **unique**, around 0.19 for **once** and between 0.09 and 0.1 for **all**. This observation means that in order to make the best decision, it is not enough to know the values for each access and the probability of emergency, one also needs to know what will be the future behaviour of the system and/or of the user.

5 Discussion - Conclusion

In this paper, we started from a very concrete problem, loosely inspired from the healthcare context, and we gradually implemented a security mechanism based on an Access Control Markov Decision Process. We have shown how changing the different parameters can have an impact on the final decision values. It is worth mentioning that the computational complexity does not change when the parameters change: in other words, it takes the same time and the same memory to compute all decision values in Table 1 than to compute those for a specific request behaviour as in Table 2. Similarly, it could be possible to create more complex reward functions, the significant computational parameter being the number of states, not the way the reward function is calculated (although these two notions can of course be related, i.e., in order to have an expressive reward function, one might need to extend the state). We believe that this approach raises the following observations and questions:

- i)* It is possible to define an access control mechanism taking into account different reward functions using linear programming for two users and two resources. What about several dozens of users and several thousand of files? It is probably not reasonable to expect to find the optimal policy at runtime, but it could be done at compile time, since we actually compute the values for *all* states.
- ii)* The ordering between the value of two decisions defines the policy (i.e., we pick the decision with the highest value), but the differential between these values is also significant, and could be used for instance to represent a notion of priority. Can we use this difference as an intuitive idea of trust?
- iii)* The future behaviour of the system has an impact: the best decision for a single access might be different whether another access will be submitted in the future, and we can calculate what this difference will be, but how reasonable is it to predict what will be the future behaviour?
- iv)* Our approach can also be used as an analysis tool, in order to understand the impact of changing one parameter, but how to define the initial parameters? How to know whether the reward (**bob, high**) is -10 and not -42?

Clearly, we probably bring more questions than answers, but we believe that using quantitative tools can help the definition of security mechanisms, in partic-

ular by providing the security engineer with a new way of thinking her security problem, and we hope to pave the way towards a generalized usage of such tools.

References

1. B. Aziz, S. N. Foley, J. Herbert, and G. Swart. Reconfiguring role based access control policies using risk semantics. *J. High Speed Networks*, 15(3):261–273, 2006.
2. R. Bellman. A markovian decision process. In. *Univ. Math. J.*, 6:679–684, 1957.
3. L. Chen and J. Crampton. Risk-aware role-based access control. In *Proceedings of 7th International Workshop on Security and Trust Management*, 2011. To appear.
4. P.-C. Cheng, P. Rohatgi, C. Keser, P. A. Karger, G. M. Wagner, and A. S. Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 222–230, Washington, DC, USA, 2007. IEEE.
5. J. Crampton and C. Morisset. An auto-delegation mechanism for access control systems. In *STM 2010*, volume 6710 of *LNCS*, pages 1–16. Springer, 2011.
6. N. N. Diep, L. X. Hung, Y. Zhung, S. Lee, Y.-K. Lee, and H. Lee. Enforcing access control using risk assessment. In *Proceedings of the Fourth European Conference on Universal Multiservice Networks*, pages 419–424. IEEE Computer Society, 2007.
7. D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. Feb. 1999.
8. H. He, Y. Shuping, and P. Wu. Security decision making based on domain partitioned markov decision process. In *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, pages 1–4, dec. 2009.
9. L. Krautsevich, A. Lazouski, F. Martinelli, and A. Yautsiukhin. Influence of attribute freshness on decision making in usage control. In *Proceedings of the 6th International Workshop on Security and Trust Management*. Springer, 2010.
10. L. Krautsevich, A. Lazouski, F. Martinelli, and A. Yautsiukhin. Risk-aware usage decision making in highly dynamic systems. In *Proceedings of The Fifth International Conference on Internet Monitoring and Protection*. IEEE, 2010.
11. L. Krautsevich, F. Martinelli, C. Morisset, and A. Yautsiukhin. Risk-based auto-delegation for probabilistic availability. In *SETOP*, pages 206–220, 2011.
12. O. P. Kreidl. Analysis of a markov decision process model for intrusion tolerance. In *Proceedings of the 2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, DSNW ’10, pages 156–161. IEEE Computer Society, 2010.
13. F. Martinelli and C. Morisset. Quantitative access control with partially-observable markov decision processes. In *CODASPY 2012*, pages 169–180. ACM, 2012.
14. I. Molloy, L. Dickens, C. Morisset, P.-C. Cheng, J. Lobo, and A. Russo. Risk-based access control decisions under uncertainty. Technical Report RC25121, IBM Watson, September 2011.
15. T. Moses. eXtensible Access Control Markup Language TC v2.0 (XACML), 2005.
16. Q. Ni, E. Bertino, and J. Lobo. Risk-based access control systems built on fuzzy inferences. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 250–260, New York, NY, USA, 2010. ACM.
17. L. Rostad and O. Eidsberg. A study of access control requirements for healthcare systems based on audit trails from access logs. In *ACSAC*, pages 175–186, 2006.
18. J. P. Singh, T. Alpcan, P. Agrawal, and V. Sharma. A markov decision process based flow assignment framework for heterogeneous network access. *Wirel. Netw.*, 16:481–495, February 2010.